CSE396 Lecture Thu. 4/8: Multitape TMs, PDAs, and General Computation

A k-tape Turing Machine (TM) has the same components $M = (Q, \Sigma, \Gamma, \delta, ␣, s, F)$ as a single-tape TM but with

$$\delta \subseteq \left(Q \times \Gamma^k\right) \times \left(\Gamma^k \times \{\mathsf{L}, \mathsf{R}, \mathsf{S}\}^k \times Q\right) .$$

It is *deterministic* (a DTM) if no two instructions share the same first two components. A DTM is "in normal form" if F consists of one state q_{acc} and there is only one other state q_{rej} in which it can halt, so that

$$\delta : (Q \setminus \{q_{acc}, q_{rej}\}) \times \Gamma^k \rightarrow \left(\Gamma^k \times \{\mathsf{L}, \mathsf{R}, \mathsf{S}\}^k \times Q\right)$$

This is read: "δ is a function from $(Q \setminus \{q_{acc}, q_{rej}\}) \times \Gamma^k$ to $\left(\Gamma^k \times \{\mathsf{L}, \mathsf{R}, \mathsf{S}\}^k \times Q\right)$." The notation then becomes $M = (Q, \Sigma, \Gamma, \delta, ␣, s, q_{acc}, q_{rej})$. All **instructions** (still also called **5-tuples** or just **tuples**) have the form

$(p, [c_1, c_2, \dots, c_k] \ / \ [d_1, d_2, \dots, d_k], [D_1, D_2, \dots, D_k], q)$

with $p, q \in Q$, $c_j, d_j \in \Gamma$, and $D_j \in \{\mathsf{L}, \mathsf{R}, \mathsf{S}\}$ $(j = 1$ to $k)$. An option I will show with $k = 2$ or 3 is to write them vertically like so:

$$\left(p, \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} / \begin{bmatrix} d_1 & D_1 \\ d_2 & D_2 \end{bmatrix}, q\right)$$

The Turing Kit allows both options, calling the latter "stacked"---IMHO it is easier to visualize. Either form gives rise to the following definition.

Definition: A **pushdown automaton** (PDA) is (equivalent to) a 2-tape Turing machine M in which every instruction has:

- $d_1 = c_1$, so that the input tape is **read-only**;
- $D_1 \neq \mathsf{L}$, so that the input tape is **one-way**; and
- $D_2 = \mathsf{L}$ only if $d_2 = ␣$, so that the tape-2 head is always on or right of the rightmost char.

The PDA is deterministic (a **DPDA**) or nondeterministic (an **NPDA**) according as M is deterministic or nondeterministic.

The third condition makes the second tape behave like a **stack**. Its head can only read the rightmost non-blank char, which is the "top" of a stack that "grows" to the right by "pushing" new chars. If the head wants to read the char to its left, the third condition makes it have to blank-out the top char, which is a "pop" move. The one cosmetic difference of using the TM notation is the need for an extra "stutter-step" to switch between "push mode" (which is when scanning the blank to the right of the topmost char) and "pop mode" (when scanning the topmost char).

Having the stay option S on the input tape is a handy coding convenience and replaces the use (IMHO, overuse) of ϵ's in the text's PDA notation in section 2.2. Another convenience is to introduce $\wedge$ as a bottom-of-stack marker on tape 2 in the first step

$$\left(s,\begin{bmatrix}\sqcup\\ \sqcup\end{bmatrix}/\begin{bmatrix}\sqcup & \mathsf{R}\\ \wedge & \mathsf{R}\end{bmatrix},q\right)$$

(and optionally, for general TMs, to introduce $\wedge$ on tape 1 as well---this is why many of my example machines in the *Turing Kit* state the convention of starting on the cell to the left of the input string). That way you can't confuse blank meaning "the stack is empty" with the blank to the right of the stack saying where you can push new chars.

[show examples using the Turing Kit, also of a non-PDA]

PDAs and CFLs

TopHat 6028

Note, incidentally, that a DFA "Is-A" DPDA that never uses its stack nor the "stay" option on tape 1, and an NFA "Is-A" NPDA. This amounts to an immediate way of seeing why every regular language is a CFL, after proving the following theorem from section 2.2 (we will mostly skip the proof):

Theorem: A language A is a CFL if and only if there is an NPDA N such that $L(N) = A$.

Proof Idea: Take a CFG $G = (V, \Sigma, \mathcal{R}, S)$ in Chomsky NF such that $L(G) = A \setminus \{\epsilon\}$. (As with the CFL Pumping Lemma, ChNF is not necessary and the text does without it, but IMHO it improves the visual understanding. If $\epsilon \in A$ we can handle that by a later patch to the code of N.) The NPDA N has just one "hub state" q (together with a helper state p for pushing) besides s and q_{acc} (and q_{rej} just to comply with the text's TM syntax). It begins with the instructions

$$\left(s,\begin{bmatrix}\sqcup\\ \sqcup\end{bmatrix}/\begin{bmatrix}\sqcup & \mathsf{R}\\ \wedge & \mathsf{R}\end{bmatrix},p\right),\ \left(p,\begin{bmatrix}c\\ \sqcup\end{bmatrix}/\begin{bmatrix}c & \mathsf{S}\\ S & \mathsf{S}\end{bmatrix},q\right)$$

which initialize the stack to hold just $\wedge$ and the grammar's start symbol S. Note that the second step leaves whatever char $c \in \Sigma$ is on the input tape alone (and if $c = \sqcup$ so that $x = \epsilon$ and we want to accept ϵ, this is where we can). A grammar rule $A \to BC$ becomes the instructions:

$$\left(q,\begin{bmatrix}c\\ A\end{bmatrix}/\begin{bmatrix}c & \mathsf{S}\\ C & \mathsf{R}\end{bmatrix},p\right),\ \left(p,\begin{bmatrix}c\\ \sqcup\end{bmatrix}/\begin{bmatrix}c & \mathsf{S}\\ B & \mathsf{S}\end{bmatrix},q\right)$$

Notice that C is pushed first, because B will be expanded next in a *leftmost* derivation and so needs to be top-of-stack. Also note that if there are multiple rules for A, they all have the same q and A, and they automatically have c for all $c \in \Sigma$ because the input tape is left alone in these moves---so N is nondeterministic. And a terminal rule $A \to c$ simply becomes the "pop" move

$$\left(q, \begin{bmatrix} c \\ A \end{bmatrix} / \begin{bmatrix} c & \text{R} \\ \llcorner\!\lrcorner & \text{L} \end{bmatrix}, q\right),$$

which also moves on to the next char on the input tape. If there is no next char, then we want to accept if and only if we just popped the last variable on the stack, which finishes off a leftmost derivation of the input x in G. So the computation should accept iff it now sees the $\wedge$ on tape 2, and this is handled by the single instruction

$$\left(q, \begin{bmatrix} \llcorner\!\lrcorner \\ \wedge \end{bmatrix} / \begin{bmatrix} \llcorner\!\lrcorner & \text{S} \\ \llcorner\!\lrcorner & \text{S} \end{bmatrix}, q_{acc}\right).$$

The **invariant** that explains why $L(N) = L(G)$ is that at any point X in a leftmost derivation $S \implies \dots \implies X \implies \dots$ of G in ChNF, the sentential form X belongs to $\Sigma^* V^*$, that is, has the form $X = uW$ where u has only terminals and W has only variables. At the same point, N has read u so far on its input tape and its stack has W in reverse---so that the first variable in W is the top-of-stack element. When the W part disappears, this means u was the whole input string x and N sees the $\wedge$, so we simultaneously have $S \implies {}^* x$ and N accepts x.

The proof in the other direction---from machine to equivalent grammar---is more complicated and is: *FYI, skim/skip* in the text. ☒

The above also contains the essence of proving the *equivalence* of two criteria for PDAs that one can find discussed in other sources: *acceptance by empty stack* and *acceptance by final state*. We can always make a PDA---nondeterministic or deterministic---do both simultaneously. One final definition and fact to wrap up the material regarded as covered in chapter 2 (it is in the first few pages of section 2.4 but just take it from here---that section is otherwise *skipped*):

Definition: A language A is a **deterministic context-free language** (**DCFL**) if there is a DPDA M such
that $L(M) = A$. The class of all DCFLs (over any given alphabet Σ) is denoted by **DCFL**.

Theorem: The complement of a DCFL is always a DCFL. That is, the class **DCFL** is closed under complementation.

Proof: Given any DPDA $M = (Q, \Sigma, \Gamma, \delta, \llcorner\!\lrcorner, s, q_{acc}, q_{rej})$, the idea is simply to do the same trick we did with DFAs: to interchange the accepting and rejecting states to make $M' = (Q, \Sigma, \Gamma, \delta, \llcorner\!\lrcorner, s, q_{rej}, q_{acc})$. We can get away with this, however, only if we first guarantee that M cannot "loop forever" with stay moves on Tape 1 and repeatedly thrashing the stack on tape 2. The details of doing so are FYI (they are gritty even with the text's PDA notation). ☒

The basic point made here becomes accentuated for general Turing machines, where we *cannot* always modify them so that they always halt. The demonstration of this is one of two main themes for the rest of the course (the other is **mapping reductions**). First, we should finally address details of computations.

Computations

An *instantaneous description* (ID), also called a *configuration*, of a Turing machine M specifies:

1. The current internal state q of M.
2. The contents $\vec{w} = w_1, \ldots, w_k$ of the k tapes, such that all else on the tapes is blank.
3. The positions $\vec{h} = h_1, \ldots, h_k$ of the heads on those tapes.

We can write $I = \langle q, \vec{w}, \vec{h} \rangle$ to denote an ID.

Write $I \vdash_M J$ if there is an instruction in δ that when executed in ID I produces ID J. For $r \geq 2$, write $I \vdash_M^r K$ if there is an ID J such that $I \vdash_M J$ and $J \vdash_M^{r-1} K$. Also write $I \vdash_M^0 I$ for all I and $I \vdash_M^* J$ if $I \vdash_M^r J$ for some r. These notions apply to nondeterministic TMs as well as DTMs.

For a single-tape TM and input x, the initial ID can be written $I_0(x) = \langle s, x, 1 \rangle$ (if we number the cells from 1) or $I_0(x) = \langle s, \wedge x, 1 \rangle$ (if we use the convention of an initial $\wedge$ in cell 0 but still number x from 1 and start up scanning the first bit rather than the $\wedge$). Yet another convention is to start in the ID $\langle s, \wedge x \$, 1 \rangle$ with a right-endmarker $\$$ too. A 1-tape TM is a *linear bounded automaton* (LBA) if δ is syntactically coded so that the only instructions involving the endmarlers have the form $(p, \wedge/\wedge, R, q)$ or $(p, \$/\$, L, q)$, so that the head always stays between $\wedge$ and $\$$.

For k-tape TMs we could use ϵ's to stand for the other tapes being blank and 1's for the other head positions, but we won't go any further into details of IDs until we hit complexity theory. An *accepting ID* has q_{acc} as its state and a *rejecting ID* has q_{rej}. Now we can formally define the language of a TM (NTMs too):

Definition: $L(M) = \left\{ x : I_0(x) \vdash_M^* I_f \text{ for some accepting ID } I_f \right\}$.

[Inserted, will cover Tue.] Given a **D**TM M and an input x, we can write "$M(x)$" to refer to the whole computation that occurs. It might never halt, in which case we write $M(x)\uparrow$ and also say $M(x)$ **diverges**. If it halts, we write $M(x)\downarrow$. If $M(x)$ halts for all inputs x, we say M is **total**. A language A is **computably enumerable** (**c.e.**) if there is a DTM M such that $L(M) = A$, and **decidable** if also M is total. When M is total we also say that M **decides** its language, while saying "M accepts A" can allow M not to be total. (There are many synonyms here, as we will see.)

Example of a 2-tape TM that decides a non-CFL, the double-word language. It also exemplifies the kind of prose description asked for on 3(b) of assignment 8.

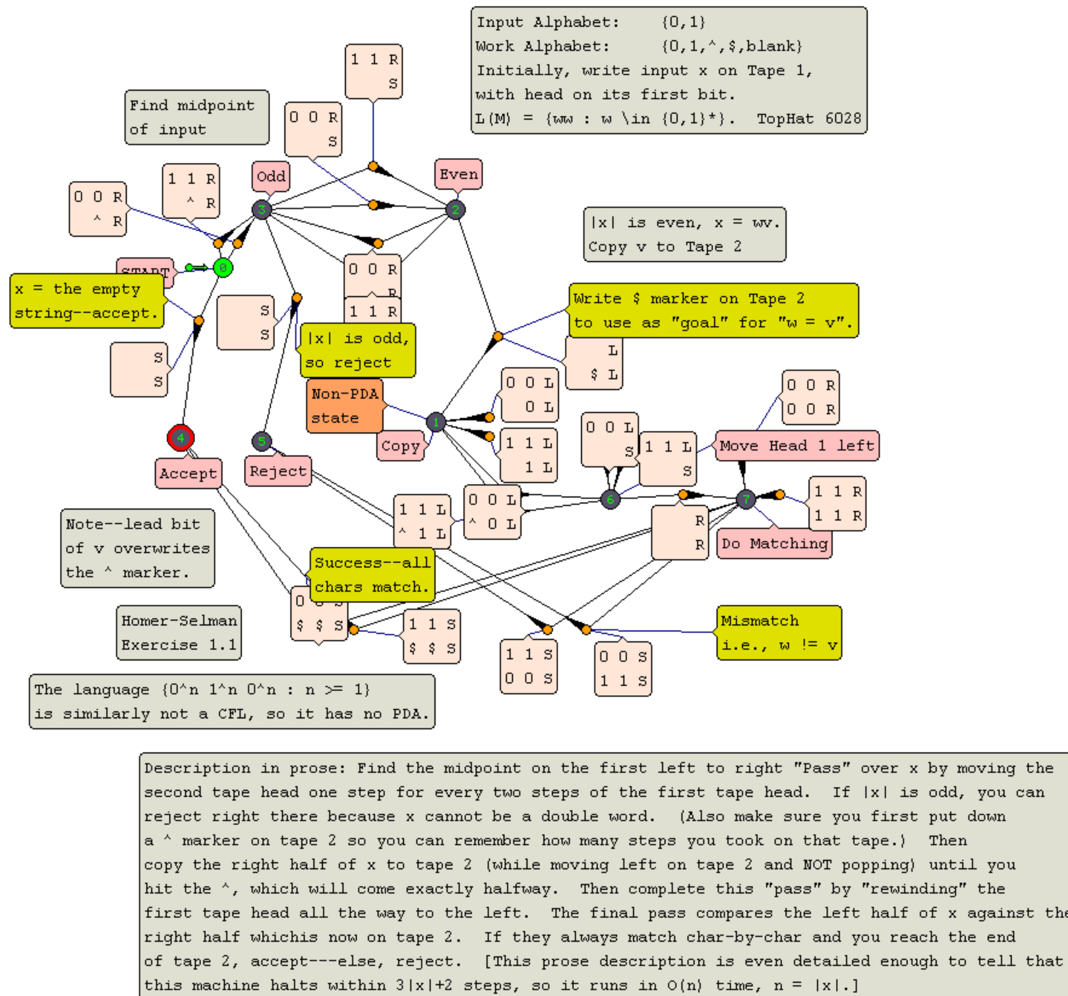

Figure 1: The description in prose at the bottom is regarded as enough to specify the particular state code with arcs and nodes. The equivalence of TMs to high-level languages (end of chapter 3 and beginning of chapter 4) justifies this theoretically.

TMs and High-Level Computation (pick up on Tuesday 4/13)

The TMs we have seen show off some basic capabilities of TMs

- copying a string
- comparing two straings
- searching for a matching (sub-)string on a tape
- arithmetic like $3n + 1$
- branching according to what char is read
- looping.

These operations suffice to build an interpreter for assembly code.

[Show Universal RAM Simulator handout, discuss the Church-Turing Thesis, and explain why it enables
machines to be specified in pseudocode from here on out. Then go into chapter 4.]